Experiments with a Real Parallel Attribute Evaluator

Martin JOURDAN, Bruno MARMOL and Didier PARIGOT
INRIA*

(*Extended Abstract*)

Abstract

We present a simple but effective method for constructing efficient attribute evaluators for the class of l-ordered attribute grammars that run on tightly-coupled (shared-memory) multiprocessor machines. We also give an account of how we implemented this method in practice. Lastly, we give some figures drawn from realistic experiments, i.e. actual implementation of parallel evaluators for meaningful AGs and their runs on meaningful source texts. The results we have obtained are quite satisfactory, since we observe a quasi-linear speedup with a number of processors varying up to a reasonable number, while the performance with one processor is already quite acceptable.

1 Introduction and Previous Work

Since compilers are heavily-used programs, it is of high interest to make them as fast as possible. One possible way of achieving this is to run them on parallel computers, provided they can exploit the available parallelism. Hand-made construction of parallel compilers is possible and has actually been done [6,7,16,19,20], but it is an error-prone process because it is hard to organize the accesses to central data structures, e.g. the program tree and the symbol table, to achieve both correctness and efficiency (cf. the "don't know yet" problem in Seshadri's works). Furthermore, methods that have been used to parallelize a compiler for a given language are not always immediately applicable to a compiler for another language. Lastly, "thinking parallel" adds up to the compiler development costs, which are already great when no good specification method is used.

In addition to reducing development costs, using a specification method allows to clearly separate the "intellectual" issues of the development, i.e. what is actually in the specification, from the more mundane implementation techniques. In particular, implementation techniques can be devised for a specification method independently from a particular application and reused for all of them. Each improvement in the implementation will then automatically benefit to all of its uses. Attribute grammars (AGs) [3,14] are a specification method that has proved quite useful for

*Authors' address: INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 LE CHESNAY Cedex. E-mail: `<name>@minos.inria.fr`

compiler construction; an evidence is that almost every modern textbook on compiler construction uses them to describe various compilation algorithms. It is clear that devising effective parallel attribute evaluation methods, i.e. implementations of AGs, is highly interesting since every compiler specified by an AG can then *automatically* and *safely* be turned into a parallel compiler. Our present work addresses this topic.

Parallel attribute evaluation has already been studied in a number of works. In the first historical implementation of AGs, the FOLDS system by Fang [5], each attribute instance is turned into a separate process that waits until all the instances it depends on are available, computes the relevant value and then signals to all the processes needing it that it's available; this method is inherently inefficient because each process only executes a few useful instructions and the overhead for managing and synchronizing the processes is much greater that what is gained through their parallel execution. Other authors use a static analysis of the AG, based on the dependencies, to statically detect *useful* parallelism. Kuiper [15] gives a general model for parallel attribute evaluators and an algorithm to detect all pairs of independent attribute instances, which can then be evaluated in parallel, but gives no particular technique therefore. Schell [17] gives algorithms to construct tree-walking evaluators that perform visits to subtrees in parallel when it is potentially useful; this is well suited for tightly-coupled architectures. Böhm and Zwanepoel [2] break up the input tree at nodes chosen by the AG author, send the resulting tree fragments to separate sites that perform attribute evaluation on each of them, with messages for synchronization and communication of attribute values; efficiency of this scheme is highly dependent on the choice of the boundary nodes. Alblas [1] propose variations of this scheme for incremental reevaluation. Kaplan and Kaiser [11] propose a much more "fine-grained" parallel incremental reevaluation method that suffers from the same problems as Fang's one. Klein and Koskimies [13] study the parallelization of one-pass evaluators. Zaring [21] gives a quite thorough analysis of the construction of parallel exhaustive and incremental evaluators for the OAG class [12]. It must be noticed here that, apart from Böhm and Zwanepoel's one, none of these works reports results of actual implementations; most of them use a simulator.

In this paper we propose a simple parallel attribute evaluation method that is similar to one of Schell and one of Zaring, and is also a simple variation of a well-known sequential evaluation method that is used in our existing AG-processing system, namely the visit sequence paradigm; it hence applies to the rather large l-ordered AG class. The parallelism it makes use of is guaranteed to be cheap because, due to the use of the total evaluation order that is characteristic of the l-ordered class, no access conflict is possible and synchronization is reduced to a minimum. It can hence be implemented very efficiently on shared-memory multi-processor machines. We also report the results of an actual implementation on a Sequent Balance machine. The test examples were typical AGs and not written especially with this evaluation technique in mind. The tests show that we achieve almost linear speedup with a number of processors varying up to a reasonable amount, whereas the results with one processor are already satisfactory.

2 The Visit Sequence Evaluation Paradigm

Since we wanted our experiment to remain as simple as possible we reused as much as possible from our existing AG-processing system, FNC-2 [9]. The sequential evaluators that FNC-2 generates are based on the visit sequences paradigm [12]; indeed, this paradigm provides for the most efficient sequential evaluators since they are completely deterministic: in addition to evaluating semantic rules, a visit sequence based evaluator performs only cheap tree-walking operations determined by the structure of the input tree. The four kinds of operations in a visit sequence are as follows:

`EVAL` s where s is a set of attribute instances in the current production, evaluates those instances using the appropriate semantic rules; the instances in the RHS of these rules are guaranteed to be available.

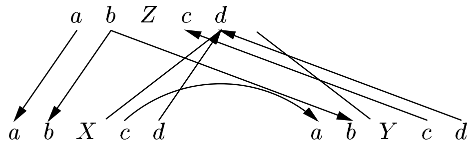
`BEGIN` i marks the start of the i-th visit to the current production (there is one visit sequence per production).

`VISIT` i, j calls for the i-th visit of the j-th son of the current production; the current node and the return address in the current visit sequence are pushed onto a stack, the j-th son becomes the current node and the production labeling this son identifies the visit sequence to use on that son; the visit number i is used to search the corresponding `BEGIN` instruction; the instruction following the latter becomes the next instruction.

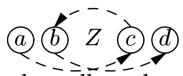
`LEAVE` i marks the end of the current (i-th) visit to the current node; if the current node is the root, evaluation is complete; otherwise, its father and the address of the next instruction to execute in the visit sequence used on the father are popped from the stack.

It is easy to see that such an evaluator can indeed be very efficiently implemented. The state variables of the interpreter are the current node, the current instruction counter and the stack of suspended visits. The latter two variables can be implemented as fields of tree nodes (one pointer to the father and one instruction counter field) if space is not at stake. An alternative to using a visit sequence interpreter is to code the visit sequences as recursive procedures, as is done in the present version of FNC-2; the visit stack then becomes implicit.

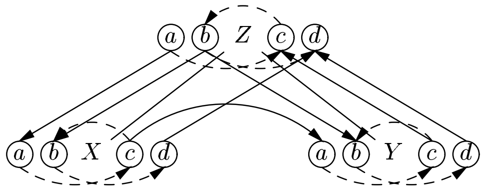
Such an evaluator can be produced only if the AG at hand is l-ordered [4]. The characteristics of such an AG is that it is possible to find, for each non-terminal, a totally-ordered partition of its attributes such that, when augmenting each production dependency graph with the corresponding edges between the attributes of the non-terminals in the RHS and LHS of the production, the graph remains acyclic. This is exemplified in Fig. 1(a, b, c). In that case, a visit sequence can be produced by topologically ordering this augmented dependency graph and enumerating the vertices as follows: inherited attributes of the LHS become `BEGIN` instructions, with increasing number of visit; inherited attributes of the RHS become `EVAL`'s; synthesized attributes of the RHS become `VISIT`'s, with increasing number of visit; and synthesized attributes of the LHS give rise to an `EVAL` followed by a `LEAVE`. Several `EVAL` instructions in a row are collapsed into a single



(a) A hypothetical production dependency graph



(b) A hypothetical totally-ordered partition on Z (the same also applying to X and Y)



(c) The augmented dependency graph is acyclic

```
BEGIN 1; EVAL {X.a}; VISIT 1, X; EVAL {Y.a};
   VISIT 1, Y; EVAL {Z.c}; LEAVE 1;
BEGIN 2; EVAL {X.b}; VISIT 2, X; EVAL {Y.b};
   VISIT 2, Y; EVAL {Z.d}; LEAVE 2.
```

(d) A possible visit sequence

Figure 1: l-orderedness test and construction of visit sequences

EVAL. A possible visit sequence for the production of Fig. 1(a) with totally-ordered partitions as in Fig. 1(b) is shown in Fig. 1(d).

3 A Simple Parallel Attribute Evaluation Method

The basic idea of our parallel evaluation method based on the visit sequence paradigm is to keep a total order for evaluating the attributes of each non-terminal (node) but allow to evaluate the attributes of different nodes, and visit those nodes, in parallel. The only difference with the construction of the sequential visit sequences is as follows: when topologically ordering an augmented dependency graph, use sequentiality when it is mandatory, but use parallelism when you come to a node with more than one successor. For instance, the second part of the visit sequence in Fig. 1(d) can be rewritten as, with obvious semantics:

```
BEGIN 2; FORK
   EVAL {X.b}; VISIT 2, X
|| EVAL {Y.b}; VISIT 2, Y
JOIN; EVAL {Z.d}; LEAVE 2.
```

The first visit cannot be parallelized because the edge $X.c \rightarrow Y.a$ mandates sequential processing.

This idea is not very bright: it consists only in taking advantage of the potential parallelism that is visible, in the most concrete sense, in the augmented dependency graphs, and that is under-exploited in sequential evaluators. It is not new either since it appears as early as 1979 in Schell's thesis [17, sec. 5.6], although only in theoretical form. Furthermore, it does not exploit all the parallelism that exists in a given AG, which requires a more thorough and more global analysis of the AG [15].However it has, in our opinion, the very important advantage that synchronization, i.e. the overhead induced by parallelism which is the possible cause of inefficiency, is reduced to a minimum since, because it is driven by the total evaluation order for the attributes of each non-terminal, the construction guarantees that:

1. all data needed by a process are computed before the process is spawned, and
2. no two processes running concurrently access the same part of the tree and, in particular, write to the same locations.

In other words, all the overhead is concentrated in the (unavoidable) **fork** and **join** instructions. So each process will run as fast as the corresponding part of the sequential evaluator. The results of section 5 will show that this technique allows to create enough parallel processes to efficiently use a multi-processor machine. But first let's see how we have actually implemented it.

4 Implementation

Our experiments were conducted on a Sequent Balance machine [18] whose features seemed, at first sight, to perfectly match our needs: multi-processor (10 processors) with shared memory; **fork/join** model of parallel computation; C language and Unix interface; local availability.

Our first job was to modify the evaluator generator of the FNC-2 system[1] to generate parallel visit sequences. In the present version, only `VISIT`s are parallelized; for instance the above sequence would be produced as:

`BEGIN 2; EVAL` $\{X.b, Y.b\}$`; FORK VISIT 2,` X $\|$ `VISIT 2,` Y `JOIN; EVAL` $\{Z.d\}$`; LEAVE 2.`

The second step was to create a new back-end (translator to C) which translated parallel visit sequences into recursive procedures with `fork` and `join` instructions; the translation of individual semantic rules was unmodified, except for the insertion of `private` or `shared` storage specifiers. It must be noted here that we make no use of the powerful space optimizations devised for the sequential case [10] because we have not (yet?) found a way to adapt them for a parallel framework; in consequence, we use the classical solution of storing all the attributes at tree nodes.

We quite early aborted the implementation of the **fork/join** model when we read in the manual that the Sequent operating system, DYNIX, implemented only heavyweight UNIX processes, in which the **fork** system call involves a complete copy of the non-shared portion of the memory image of the parent process [18]; the corresponding time penalty (of the order of 50 ms) was intolerable, given that the processes we generate are quite small. Our solution is hence to replace the recursive procedure with **fork/join** model by a purely interpretive model using our own specialized scheduler. Let us dive into the details of the operation.

In addition to the classical fields (pointers to the sons, production number, attribute values), each tree node contains the following fields: a pointer to its father; a field for holding the return address in the visit sequence of the father, for `VISIT` and `LEAVE` instructions; a "join counter"

[1] In the FNC-2 system the construction of an evaluator from an AG is performed in two steps: we first produce (abstract) visit sequences, then we translate them into the chosen implementation language. Please refer to [9] for more details.

```
doforever
    wait for something in the job queue;
    pick the first job from the queue;
    node ← job.node;
    VSC ← job.VSC;
    done ← false;
    repeat
        case VS[VSC] is { VS is the global array containing the visit sequences }
        EVAL s:     call the relevant evaluation procedure, with node as unique parameter;
                    VSC ← next(VSC);
        VISIT i, j: node ← node.son[j];
                    node.ret-addr ← next(VSC);
                    VSC ← choose-VSC(node.prod, i);
        LEAVE i:    if node = root then terminate() end if;
                    VSC ← node.ret-addr;
                    node ← node.father;
        FORK:       node.join-counter ← number of following parallel branches;
                    foreach parallel branch do
                        enqueue a job descriptor with job.node = node
                                and job.VSC = first instruction of the branch
                    endfor;
                    done ← true; { fall back to the scheduler }
        ||:         VSC ← address of the JOIN instruction closing the current parallel block;
        JOIN:       lock(node.join-lock);
                    decrement(node.join-counter);
                    jc ← node.join-counter;
                    unlock(node.join-lock);
                    if jc = 0 then { JOIN is complete, continue with the visit sequence }
                        VSC ← next(VSC)
                    else { fall back to the scheduler }
                        done ← true
                    end if
        end case
    until done
enddo
```

Figure 2: Algorithm for the parallel visit sequence interpreter

and an associated lock variable. Note that an alternative is to store this information in the job descriptors described below.

In addition to the tree and attributes, the global variables are the visit sequences themselves, with `FORK` and || and `JOIN` instructions as above, and a job queue with associated lock variable. A job descriptor in this queue contains a pointer to some node and a visit sequence counter pointing to some `BEGIN` instruction in the visit sequence for this node.

The local variables of each process are a pointer to the current node (*node*) and a visit sequence counter (*VSC*) pointing to the current instruction in the visit sequence. Each process executes the algorithm in Fig. 2, which is a modified visit sequence interpreter. We omitted the details of initialization and termination, those of the queue management (locking, etc.), and those of the representation of the visit sequences (e.g. the *next* function).

The reader may wonder why, in a `VISIT` instruction, the return address in the visit sequence of the father is stored in the visited son rather than in the father. The reason is that the construction

AG	check	term
# non-terminals	11	11
# productions	21	21
# semantic rules	100	69
# attr./NT (ave.)	2.45	1.73
# visits/NT (max.)	2	1
# visits/NT (ave.)	1.57	1
# parallel blocks	9	9
# branches/block (ave.)	2.55	2.44

Table 1: Characteristics of AGs 1 and 2

text	1	2	3	4
characters	6206	11395	11585	22725
lines	399	721	731	1437
# procedures	35	70	2×35	2×70
speedup with 6 procs. (`check`)	3.42	3.90	4.24	5.17
speedup with 6 procs. (`term`)	2.13	2.24	2.42	2.71

Table 2: Characteristics of the source texts

guarantees that any node is being visited at most once at a given time whereas a same father may spawn more than one visit at the same time, through `FORK` instructions, so it is not safe to store this return address in the father.

Note that the semantics of the `EVAL`, `VISIT` and `LEAVE` instructions is the same as in the sequential case, even if their implementation is slightly different. Lastly, an obvious improvement is to have the `FORK` instruction enqueue a job descriptor for all but one of the parallel branches and keep the last one for execution by the current process.

5 Results of experiments

Parallel evaluators were constructed for two AGs whose characteristics are summarized in Table 1. They are those of the *simproc* toy-language used as example in the FNC-2 Reference Manual [8]. The first one describes the verification of the contextual constraints, the second one describes the construction of an intermediate representation. These AGs were not modified at all for the experiment. We believe that they are typical of, although much simpler than, those that could be written for compiler construction applications. Furthermore, using larger AGs would lead to results even better that those presented below since more opportunities for parallelism could be exploited.

These AGs were run on a number of source texts whose characteristics are summarized in Table 2. The texts were produced by assembling a varying number of copies of the factorial procedure in either a flat list (first two ones) or by grouping them into two enclosing procedures. More realistic source texts of the same size would probably lead to more complex trees, thus exhibiting more opportunities for parallelism.

The speedup figures were computed by measuring the real (wall-clock) running time of the

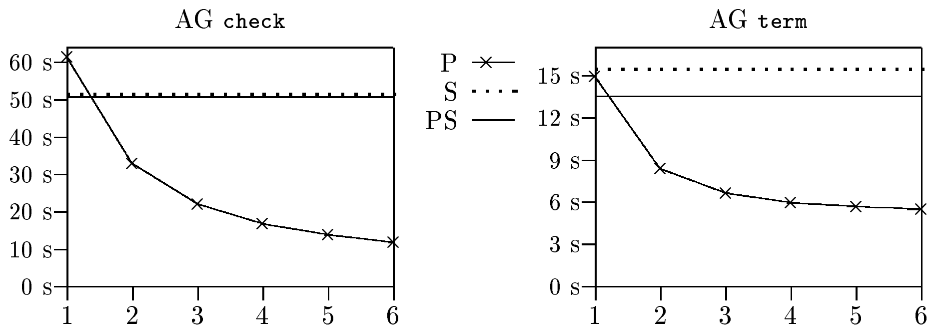


Figure 3: Detailed results for text # 4. Horizontal axis is the number of processors, vertical axis is (real) running time. P is the parallel interpreter, S is a sequential interpreter, PS is the parallel interpreter driven by sequential visit sequences.

evaluators on a varying number of processors. It must be noted that this time does not include parsing and tree construction but only pure attribute evaluation. We believe that these figures are quite satisfactory, given the simplicity of our evaluation method. Furthermore, as said above, they are sort of a lower bound: we expect measures on more realistic examples to be even better. Fig. 3 gives more detailed results for our largest source text; these figures are the most significant ones and will be the basis for the following discussion.

The performance of the parallel evaluator (P) on the `check` AG is a perfect example of successful parallelization: on one processor it is comparable (only 20% slower) to a sequential evaluator (S), which means that the overhead caused by the parallelization is quite small, and adding more processors leads to quasi-linear speedup. The efficacy, i.e. the quotient of the speedup by the number of processors, smoothly decreases from 93% with 2 processors to 86% with 6 of them; these are very good figures. The parallel evaluator running on only two processors is already 35% faster than the sequential one.

The performance on the `term` AG is much less good. This is because `check` performs much more data traversal, i.e. pure computation, than data creation (allocation), whereas it is the opposite for `term`, which merely creates a new representation of the input program that is quasi-isomorphic to the input tree. The shape of the curve for `term` clearly shows that the shared-memory allocator is the bottleneck here: the efficacy with two processors is almost the same as for `check` but the floor is reached with four processors and adding more of them decreases only marginally the running time. It must be noticed here that, right from the beginning, we had to replace the standard *shmalloc* allocator by a much lighter version that directly called *shsbrk* to observe *any* speedup at all. The fact that the sequential evaluator (S) runs a bit slower on `term` than the parallel evaluator driven by sequential visit sequences (i.e. with no `FORK`, etc.) can be explained by the fact that, on `VISIT` instructions, the latter stores the pointer to the father and the return address in the (pre-allocated) tree while the former stores them in a separate stack that it must explicitly manage; this "overhead" is less visible in more computation-bound AGs such as `check`.

Note to the reviewers and program committee: Lack of time before the deadline for submission prevented us from performing more thorough experiments with more AGs and more source texts. We are doing them now, in particular with the same AGs as in [9], and we'll report the results in the final version of the paper.

6 Conclusion and Future Work

We have presented in this paper a simple but effective parallel attribute evaluation method that applies to the large l-ordered class. The parallelism it makes use of is guaranteed to be cheap because synchronization costs are reduced to a minimum. Its implementation on shared-memory multi-processor machines is easy. Our practical experiments, although not numerous enough, already show that this method is able to fully exploit the power of the machine it runs on. We have thus proved that parallel attribute evaluation is quite possible and is a highly valuable method for constructing parallel compilers at low cost.

In addition to performing more experiments, we intend to pursue this work in two directions:

- restrict parallelism by a "less blind" static analysis of the AG, i.e. parallelize only at well-chosen nodes (as in [2] but more automatically); this would lead to larger parallel processes and smaller overhead;
- devise specific space optimization techniques for this evaluation method.

Acknowledgements Thanks to Luc Maranget and Robert Ehrlich for pointing to *shmalloc* as a possible (and actual) anti-concurrent bottleneck.

References

1. Alblas, H. *Concurrent Incremental Attribute Evaluation*. In *Attribute Grammars and their Applications (WAGA)*, P. Deransart and M. Jourdan, Eds. Lect. Notes in Comp. Sci., vol. 461, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 343–358.
2. Böhm, H-J. and Zwanepoel, W. *Parallel Attribute Grammar Evaluation*. In *7th Int. Conf. on Distributed Computing Systems*, R. Popescu-Zeletin, G. Le Lann and K. H. Kim, Eds. Sept. 1987, 347–354.
3. Deransart, P., Jourdan, M. and Lorho, B. *Attribute Grammars: Definitions, Systems and Bibliography*. Lect. Notes in Comp. Sci., vol. 323, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.
4. Engelfriet, J. and Filé, G. Simple Multi-Visit Attribute Grammars. *J. Comput. System Sci. 24*3 (June 1982), 283–314.
5. Fang, I. FOLDS, a Declarative Formal Language Definition System. Comp. Sc. Dept., Stanford Univ., PhD thesis, report STAN-CS-72-329, Dec. 1972.
6. Frankel, J. L. The Architecture of Closely-coupled Distributed Computers and their Language Processors. Dept. of Applied Maths., Harvard Univ., PhD thesis, 1983.

7. Gross, T., Zobel, A. and Zolg, M. Parallel Compilation for a Parallel Machine. *ACM SIGPLAN Notices 24*7 (July 1989), 91–100.

8. Jourdan, M. and Parigot, D. *The FNC-2 System User's Guide and Reference Manual* release 0.4. INRIA, Rocquencourt, Feb. 1989.

9. Jourdan, M., Parigot, D., Julié, C., Durin, O. and Le Bellec, C. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. *ACM SIGPLAN Notices 25*6 (June 1990), 209–222.

10. Julié, C. and Parigot, D. *Space Optimization in the FNC-2 Attribute Grammar System*. In *Attribute Grammars and their Applications (WAGA)*, P. Deransart and M. Jourdan, Eds. Lect. Notes in Comp. Sci., vol. 461, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 29–45.

11. Kaplan, S. M. and Kaiser, G. E. *Incremental Attribute Evaluation in Distributed Language-based Environments*. In *5th ACM Symp. on Principles of Distributed Computing*. Aug. 1986, 121–130.

12. Kastens, U. Ordered Attribute Grammars. *Acta Inform. 13*3 (1980), 229–256.

13. Klein, E. and Koskimies, K. *Parallel One-pass Compilation*. In *Attribute Grammars and their Applications (WAGA)*, P. Deransart and M. Jourdan, Eds. Lect. Notes in Comp. Sci., vol. 461, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 76–90.

14. Knuth, D. E. Semantics of Context-free Languages. *Math. Systems Theory 2*2 (June 1968), 127–145.

15. Kuiper, M. F. and Swierstra, S. D. *Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism*. In *Attribute Grammars and their Applications (WAGA)*, P. Deransart and M. Jourdan, Eds. Lect. Notes in Comp. Sci., vol. 461, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 61–75.

16. Lipkie, D. E. A Compiler Design for Multiple Independent Processor Computers. Dept. of Comp. Sc., Univ. of Washington, PhD thesis, Seattle, WA, 1979.

17. Schell, R. M. Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment. Dept. of Comp. Sc., Univ. of Illinois at Urbana-Champaign, PhD thesis, report UIUCDCS-R-79-0958, Feb. 1979.

18. Sequent Computer Systems, Inc. Balance Guide to Parallel Programming. 1003-43967 Rev. A, Nov. 1986.

19. Seshadri, V., Wortman, D. B., Junkin, M. D., Weber, S., Yu, C. P. and Small, I. Semantic Analysis in a Concurrent Compiler. *ACM SIGPLAN Notices 23*7 (July 1988), 233–239.

20. Vandevoorde, M. T. Parallel Compilation on a Tightly-coupled Multiprocessor. DEC Systems Research Center, SRC report 26, Palo Alto, CA, Mar. 1988.

21. Zaring, A. K. Parallel Evaluation in Attribute Grammar-based Systems. Dept. of Comp. Sc., Cornell Univ., PhD thesis, Tech. Rep. 90-1149, Aug. 1990.